Programowanie 2009 Programming 2009

Lista zadań nr 5 Problem set no. 5

Na zajęcia 31 marca – 1 kwietnia 2009 Due April 1, 2009

Grupy zasadnicze Basic groups

Zadanie 1 (1 pkt). Zaprogramuj w Haskellu funkcję

Problem 1 (1 p). Define in Haskell a function

```
roots :: (Double, Double, Double) -> [Double]
```

wyznaczającą listę miejsc zerowych trójmianu kwadratowego o podanych współczynnikach. *Wskazówka:* typ `Double` należy do klasy `Ord`, zdefiniowano więc dla niego metodę

which finds a list of roots of a quadratic function with given coefficients. *Hint:* the type `Double` belongs to the class `Ord`, so the method

```
compare :: Double -> Double -> Ordering
```

gdzie

is defined for it, where

```
data Ordering = LT | EQ | GT deriving (Eq, Ord, Enum, Read, Show, Bounded)
```

W preludium standardowym zdefiniowano też funkcję

A function

```
sqrt :: (Floating a) => a -> a
```

a typ `Double` należy do klasy `Floating`.

is also defined in the standard prelude and the type `Double` belongs to the class `Floating`.

Zadanie 2 (1 pkt). Niech

Problem 2 (1 p). Let

```
data Roots = No | One Double | Two (Double, Double) deriving Show
roots :: (Double, Double, Double) -> Roots
```

Zaprogramuj tę funkcję. Czy jest lepsza niż poprzednia? Podaj argumenty za i przeciw. Skomentuj następnie funkcje o sygnaturach:

Define this function. Is it better than the previous one? Give some pros and cons. Then comment functions having the following signatures:

```
roots :: Double -> Double -> Double -> [Double]
roots :: [Double] -> [Double]
```

Zadanie 3 (1 pkt). Nie korzystając z faktu, że typ `Integer` należy do klasy `Show` zaprogramuj funkcję

Problem 3 (1 p). Not using the fact that the `Integer` type belongs to the `Show` class define a function

```
integerToString :: Integer -> String
```

Wskazówka: wykorzystaj funkcję

Hint: use the function

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

z modułu `List` daną wzorem

from the `List` module, defined by the formula

```
unfoldr f b =
   case f b of
      Nothing    -> []
      Just (a,b) -> a : unfoldr f b
```

gdzie

where

```
data Maybe a = Nothing | Just a deriving (Eq, Ord, Read, Show)
```

Moduł `Char` udostępnia funkcję

The `Char` module exports the function

```
intToDigit :: Int -> Char
```

preludium standardowe oferuje funkcję

the standard prelude offers the function

```
fromEnum :: (Enum a) => a -> Int
```

a typ `Integer` należy do klasy `Enum`.

and the `Integer` type belongs to the `Enum` class.

Zadanie 4 (1 pkt). Zdefiniuj listę

Problem 4 (1 p). Define a list

```
nat2 :: [(Integer,Integer)]
```

zawierającą wszystkie pary nieujemnych liczb całkowitych w kolejności określonej przez — znany z dowodu tw. Cantora — porządek

that contains all pairs of nonnegative integers ordered by the relation known from the proof of Cantor theorem:

$$(x_1, y_1) \prec (x_2, y_2) \iff x_1 + y_1 < x_2 + y_2 \vee (x_1 + y_1 = x_2 + y_2 \wedge x_1 < x_2)$$

tj. takiej, że

i.e., such that

```
nat2 = [(0,0),(0,1),(1,0),(0,2),(1,1),(2,0),(0,3),(1,2),(2,1),(3,0),...]
```

Wskazówka: definicja powinna zmieścić się w jednym wierszu długości mniejszej niż 45 znaków. Zauważ, że suma współrzędnych punktów leżących na tej samej przekątnej jest stała.

Hint: the definition should fit in one line shorter than 45 characters. Notice that the sum of coordinates of points laying on on the same diagonal is constant.

Zadanie 5 (1 pkt). Zaprogramuj w Haskellu algorytm sortowania przez proste wstawianie.

Problem 5 (1 p). Define in Haskell the insertion sort algorithm.

Zadanie 6 (1 pkt). Zaprogramuj w Haskellu algorytm sortowania przez proste wybieranie.

Problem 6 (1 p). Define in Haskell the selection sort algorithm.

Zadanie 7 (1 pkt). Zaprogramuj w Haskellu algorytm Quicksort.

Problem 7 (1 p). Define in Haskell the Quicksort algorithm.

Grupy rozszerzone

Extended groups

Zadanie 1 (1 pkt). Zaprogramuj w Haskellu algorytm Mergesort.

Problem 1 (1 p). Define in Haskell the Mergesort algorithm.

Zadanie 2 (1 pkt). Wykorzystaj funkcję

Problem 2 (1 p). Use the function

```
merge :: Ord a => [a] -> [a] -> [a]
```

do rozwiązania problemu 2-3-5 Dijkstry: zdefiniuj nieskończony rosnący ciąg liczb całkowitych

to solve the Dijkstra 2-3-5 problem: define an infinite increasing sequence of integers

```
d235 :: [Integer]
```

zawierający liczbę 1 i taki, że jeśli n jest elementem tego ciągu, to są nimi też $2n$, $3n$ i $5n$. *Uwaga:* w odróżnieniu od funkcji wykorzystanej w algorytmie Mergesort, ta funkcja `merge` powinna usuwać duplikaty.

containing 1 and such that if n belongs to that sequence, then so do $2n$, $3n$ and $5n$. *Remark:* in contrast to the function used in the Mergesort algorithm, this function `merge` should remove duplicates.

Zadanie 3 (1 pkt). Rozważmy binarne drzewa poszukiwań

Problem 3 (1 p). Consider binary search trees

```
data Tree a = Node (Tree a) a (Tree a) | Leaf
```

Zaprogramuj w Haskellu następujące funkcje

Define the following functions in Haskell

```
insert :: Ord a => a -> Tree a -> Tree a
flatten :: Tree a -> [a]
treeSort :: Ord a => [a] -> [a]
```

Zadanie 4 (1 pkt). Binarne drzewa poszukiwań służą jako efektywne reprezentacjami zbiorów skończonych. Przyjmuje się wówczas, że drzewa nie zawierają duplikatów. Zaprogramuj następujące operacje:

Problem 4 (1 p). Binary search trees serve as effective representations of finite sets. It is assumed that they do not contain duplicates. Define the following operations:

```
type Set a = Tree a
empty :: Set a
isEmpty :: Ord a => Set a -> Bool
singleton :: Ord a => a -> Set a
fromList :: Ord a => [a] -> Set a
union :: Ord a => Set a -> Set a -> Set a
intersection :: Ord a => Set a -> Set a -> Set a
member :: Ord a => a  -> Set a -> Bool
```

Zadanie 5 (1 pkt). Zbiory, także nieskończone, można reprezentować w postaci ich funkcji charakterystycznych:

Problem 5 (1 p). Sets, even infinite ones, can be represented by their characteristic functions:

```
newtype FSet a = FSet (a -> Bool)
```

Zdefiniuj następujące operacje:

Define the following operations:

```
empty :: FSet a
singleton :: Ord a => a -> FSet a
fromList :: Ord a => [a] -> FSet a
union :: Ord a => FSet a -> FSet a -> FSet a
intersection :: Ord a => FSet a -> FSet a -> FSet a
member :: Ord a => a -> FSet a -> Bool
```

Zadanie 6 (1 pkt). Kopiec n-elementowy to drzewo binarne, w którym k-ty element ($1 \leq k \leq n$) znajduje się w wierzchołku na końcu ścieżki prowadzącej od korzenia poprzez wierzchołki zgodnie z dwójkowym rozwinięciem liczby k od najmniej znaczącej cyfry do cyfry poprzedzającej najbardziej znaczącą jedynkę. Cyfra 0 w rozwinięciu oznacza, że wybieramy lewego syna, 1 zaś — że prawego. Np. 6. element kopca znajduje się w wierzchołku będącym prawym synem lewego syna. Kopiec spełnia też w każdym wierzchołku *warunek kopca*: etykiety synów są nie mniejsze niż etykieta danego wierzchołka. Niech

Problem 6 (1 p). An n-element heap is a binary tree, in which the k-th element ($1 \leq k \leq n$) is placed in a node at the end of a path leading from the root of the tree through nodes according to the binary representation of the number k from the least significant digit to a digit that precedes the most significant one. The digit 0 means that we choose the left son, 1 — that the right. *E.g.*, the 6-th element of a heap is put in the node which is the right son of the left son of the root. The heap also satisfies in every node *the heap condition*: the labels of the sons are not smaller than the label of the node. Let

```
newtype Heap a = Heap (Int, Tree a))
```

gdzie konstruktor `Heap` przechowuje kopiec i liczbę jego elementów. Zaprogramuj w Haskellu następujące funkcje

where the constructor `Heap` stores a heap and the number of its elements. Define the following functions in Haskell

```
insert :: Ord a => a -> Heap a -> Heap a
getMin :: Ord a => Heap a -> Maybe (a, Heap a)
heapSort ::  Ord a => [a] -> [a]
```

Zadanie 7 (1 pkt). Rozważmy funkcję

Problem 7 (1 p). Consider the function

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Dla jakich par funkcji `f` i `f'` zachodzi równość

For which pairs of functions `f` i `f'` the following equation holds?

```
unfoldr f' (foldr f z xs)  =  xs
```

Grupa zaawansowana

Advanced group

Zadanie 1 (2 pkt). Rozważmy maszynę RAM przechowującą w pamięci tylko do odczytu jednokierunkową listę wiązaną długości n. Maszyna ma dodatkowo m komórek pamięci do odczytu i zapisu. Zamierzamy odwiedzić elementy tej listy w kolejności od ostatniego do pierwszego. Łatwo zauważyć, że jeśli mamy do dyspozycji $m = \Omega(n)$ dodatkowych komórek pamięci, to wystarczy odwrócić listę i przejść odwróconą listę w naturalnej kolejności. Czas działania takiego algorytmu wynosi $O(n)$. Łatwo też znaleźć algorytm, który przechodzi listę do tyłu w stałej pamięci, ale w czasie kwadratowym. Pokaż, że dla dowolnego ustalonego $k \in \mathbb{N}$ można przejść listę pod prąd w czasie $O(n^{1+1/k})$ używając $m = O(1)$ dodatkowych komórek pamięci. *Wskazówka:* wymyśl wpierw algorytm, który zużywa $m = O(\sqrt{n})$ komórek pamięci i działa w czasie $O(n^{3/2})$.

Problem 1 (2 p). Consider a RAM machine storing a singly linked list of length n in a read-only memory. The machine has m additional memory cells for reading and writing. We are going to visit all elements of the list from the last to the first. It is easy to observe that if we have $m = \Omega(n)$ additional memory cells it suffices to reverse the list and visit all elements of the reversed list in the natural order. The running time of this algorithm is $O(n)$. It is as well easy to find an algorithm that traverses a list backward in constant space but in quadratic time. Show that for any constant $k \in \mathbb{N}$ it is possible to walk through a list backwards in time $O(n^{1+1/k})$ using $m = O(1)$ additional memory cells. *Hint:* first devise an algorithm that uses $m = O(\sqrt{n})$ memory cells and runs in $O(n^{3/2})$ time.

Zadanie 2 (2 pkt). Zaprogramuj algorytm Mergesort w Prologu i Haskellu. W Prologu użyj list różnicowych, by ograniczyć kopiowanie i śmiecenie. Jak można zmniejszyć śmiecenie w Haskellu? Udowodnij, że pesymistyczny czas działania algorytmu oraz łączna liczba alokowanych komórek pamięci wynoszą $O(n \log n)$. Ile komórek pamięci jest jednocześnie żywych (patrz następne zadanie)? Jaka jest głębokość rekursji.

Problem 2 (2 p). Define the Mergesort algorithm in Prolog and in Haskell. Use difference lists in Prolog to mitigate copying and creation of garbage. How can you mitigate creation of garbage in Haskell? Prove that the pessimistic running time of the algorithm and the total number of allocated memory cells are $O(n \log n)$. How many memory cells are simultaneously alive (see next problem). What is the recursion depth?

Zadanie 3 (2 pkt). Aby opisać zarządzanie pamięcią w językach wysokiego poziomu (tutaj „język wysokiego poziomu" to język w którym pamięć jest zarządzana automatycznie) rozważa się zwykle pamięć o adresach z pewnego nieskończonego abstrakcyjnego zbioru $\mathbb{A}$. Na adresach wolno wykonywać jedynie operacje

Problem 3 (2 p). To describe the memory management in high-level languages (here by a "high-level language" we mean a language in which memory is managed automatically) one usually considers a memory with addresses taken from some infinite abstract set $\mathbb{A}$. The only permitted operations on addresses are

$$\texttt{new} \;:\; (\mathbb{N} \cup \mathbb{A})^* \to \mathbb{A}$$
$$\texttt{lookup} \;:\; \mathbb{A} \times \mathbb{N} \to (\mathbb{N} \cup \mathbb{A})$$

gdzie $\texttt{new}(x_1, \ldots, x_n)$ rezerwuje nową komórkę pamięci i inicjuje ją krotką $(x_1, \ldots, x_n)$, zaś $\texttt{lookup}(a, i)$ ujawnia i-ty składnik krotki przechowywanej w komórce o adresie a. (W językach imperatywnych mamy dodatkowo operację

where $\texttt{new}(x_1, \ldots, x_n)$ allocates a fresh memory cell and initializes its contents with a tuple $(x_1, \ldots, x_n)$, and $\texttt{lookup}(a, i)$ returns the i-th component of a tuple stored in a memory cell addressed with a. (In imperative languages one also has the operation

$$\texttt{update} \;:\; \mathbb{A} \times (\mathbb{N} \cup \mathbb{A})^* \to ()$$

zmieniającą zawartość komórki pamięci.) Poza pamięcią maszyna posiada też ustaloną liczbę rejestrów. Pamięć i rejestry można przedstawić w postaci grafu skierowanego, którego wierzchołkami są rejestry i komórki pamięci, a krawędzie prowadzą od rejestrów i komórek zawierających adresy do komórek o tych adresach. Komórka pamięci jest „żywa", jeśli jest osiągalna w tym grafie z pewnego rejestru maszyny. W przeciwnym razie komórka jest „martwa". Maszynę z abstrakcyjnymi adresami tłumaczymy do „rzeczywistej" maszyny RAM, której komórki pamięci są adresowane liczbami naturalnymi. Udowodnij, że jeśli pewien algorytm działa na maszynie z abstrakcyjnymi adresami w czasie t i w każdym momencie jego działania jest nie więcej niż m żywych komórek pamięci, to jego wykonanie na maszynie RAM można tak przeplatać operacjami usuwania nieużytków, by całkowity czas działania wyniósł $O(t)$, a w każdym momencie obliczenia potrzebna pamięć maszyny RAM nie przekraczała $O(m)$. Zatem analizując złożoność algorytmów możemy przyjąć, że komputer dysponuje nieograniczoną pamięcią, a nieużytki nie są usuwane. Takie założenie nie zmienia bowiem klasy złożoności algorytmu. Np. algorytm Mergesort zaprogramowany w Prologu lub Haskellu rezerwuje $\Theta(n \log n)$ komórek pamięci (w odróżnieniu od programu imperatywnego, który działa w pamięci $\Theta(n)$), ale w każdej chwili jego działania żywe jest jedynie $O(n)$ komórek. Można go wiec przetłumaczyć na maszynę RAM działającą w pamięci $O(n)$ i w czasie $O(n \log n)$, jak w przypadku imperatywnym.

that modifies the contents of memory cells.) Besides memory the machine has also a fixed number of registers. The memory and registers can be presented in the form of a directed graph, in which vertices stand for memory cells and registers, and edges lead from register and memory cells containing addresses to cells having those addresses. A memory cell is "alive" if it is reachable from some register in this graph. Otherwise the memory cell is "dead". The abstract pointer machine is translated to a "real" RAM machine, in which memory cells are addressed with natural numbers. Prove that if an algorithm runs on the abstract pointer machine in time t and in every moment of its computation the number of alive memory cells does not exceed m, then its execution on the RAM machine can be interleaved with operations of garbage collection so that the total running time is $O(t)$, and the amount of needed memory cells does not exceed $O(m)$ in every moment of the computation. Hence if we analyze the complexity of algorithms we can assume that an infinite memory is available for a computer and garbage is not removed, because this assumption does not change the complexity class of an algorithm. For example the Mergesort algorithm programmed in Prolog or Haskell allocates $\Theta(n \log n)$ memory cells (in contrast to an imperative program which runs in $\Theta(n)$ memory), but in every moment of computation there are only $O(n)$ alive memory cells. Hence it is possible to translate such program to the RAM machine that runs in $O(n)$ memory and $O(n \log n)$ time, as in the imperative case.

Zadanie 4 (2 pkt). Zaprogramuj w Prologu algorytm sortowania list, który działa w czasie oczekiwanym $O(n \log n)$ i alokuje łącznie $O(n)$ komórek pamięci. Czy potrafisz ten algorytm zaprogramować w *czystym* Prologu (bez odcięć, porównywania wskaźników itp.)? Czy potrafisz zaprogramować algorytm, który działa w pesymistycznym czasie $O(n \log n)$ i alokuje $O(n)$ komórek pamięci? Czy możesz to samo zrobić w Haskellu?

Problem 4 (2 p). Define in Prolog a list sorting algorithm that runs in $O(n \log n)$ expected time and allocates $O(n)$ memory cells in total. Can you define such an algorithm in *pure* Prolog (without cuts, pointer comparison etc.)? Can you define an algorithm with pessimistic running time $O(n \log n)$ which allocates $O(n)$ memory cells? Can you do the same in Haskell?